Safe Exploration for Dynamic Computer Systems Optimization

by

Hyunji Kim

B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author . . . .
Department of Electrical Engineering and Computer Science
May 6, 2022

Certified by . . . .
Michael J. Carbin
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . .
Yi Ding
Postdoctoral Associate
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Safe Exploration for Dynamic Computer Systems Optimization

by

Hyunji Kim

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Modern computer systems need to execute under strict safety constraints (e.g., power limit), but doing so often conflicts with their ability to deliver high performance (i.e., minimal latency). To meet these conflicting goals, prior work uses machine learning to automatically tune hardware resources such that system executions meet safety constraints optimally. Such solutions monitor past system executions to learn the system's behavior under different hardware resource allocations before dynamically tuning resources to optimize the application execution. However, system behavior can change significantly between different applications and even different inputs of the same applications. Hence, models learned using data collected a priori are often suboptimal and violate safety constraints when used with new applications and/or inputs.

To address this limitation, I introduce the concept of an *execution space*, which is the cross product of hardware resources, input features, and applications. Thus, a *configuration* is defined as a tuple of hardware resources, input features, and application. To dynamically and safely allocate hardware resources from the execution space, I present SCOPE [1], a resource manager that leverages a novel safe exploration framework. SCOPE operates iteratively, with each iteration (i.e., reallocation) having three phases: monitoring, safe space construction, and objective optimization. To construct a safe set with high coverage (i.e., a high number of safe configurations in the predicted safe set), SCOPE introduces a locality preserving operator so that SCOPE's exploration will rarely violate the safety constraint and have small magnitude violations if it does. I evaluate SCOPE's ability to deliver improved latency while minimizing power constraint violations by dynamically configuring hardware while running a variety of Apache Spark applications. Compared to prior approaches that minimize power constraint violations, SCOPE consumes comparable power while improving latency by up to 9.5×. Compared to prior approaches that minimize latency, SCOPE achieves similar latency but reduces power constraint violation rates by up to 45.88×, achieving almost zero safety constraint violations across all applications.

[1] **S**afe **C**onfiguration **O**ptimization for **P**erformance and **E**fficiency

Thesis Supervisor: Michael J. Carbin
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Yi Ding
Title: Postdoctoral Associate

Acknowledgments

First and foremost, I would like to thank my thesis advisor, Professor Michael Carbin, for welcoming me into the Programming Systems Group and supporting me throughout my MEng journey. I am so grateful for all of his expertise, feedback and considerations that allowed me to have such an enriching MEng experience.

To my mentor and supervisor, Yi Ding, I would like to say thank you for her patience and guidance during our countless discussions and struggles. Yi has always been there to answer any questions or offer feedback, and I am deeply appreciative of the time and passion she dedicated into the project.

I am especially grateful to have had amazing collaborators on this project. Thanks to Yi for her help in the development of the ideas for the whole project, motivational examples, and experimentation as well as her help in writing and editing all sections of this thesis. Thanks to Ahsan Pervaiz for helping me set up the data collection process and spending numerous hours debugging errors, that was crucial for experimenting.

Additionally, I would like to express my gratitude to Professor Henry Hoffmann for continuously offering suggestions and providing feedback on the project. I would also like to thank everyone in the Programming Systems Group for all their kind help, encouragements, and pleasant interactions; special thanks to Alex Renda and Tian Jin for providing detailed feedback for the project. I feel so lucky to have had the chance to meet, work with and learn from everyone throughout my MEng journey.

Of course, I cannot leave out my family and friends. I am grateful to all of my family and friends, for without their love and support, I would not be where I am today.

Last but not least, I am thankful to have been supported by a Meta Research Award for my MEng thesis work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A modern computer system needs to meet conflicting goals—for example, minimizing latency while meeting some safety constraint (e.g., power limit)—in the face of dynamic changes in its application and environment. To do so, hardware architects expose a wide variety of resources for the system to manage [1, 2, 3], where each type of hardware resource is controlled by a *hardware parameter* and all possible allocations of hardware resources constitute a *resource space*.

When managing these resources, the system needs to meet its safety constraints. Rare and small magnitude violations can be tolerable if they do not dramatically degrade system performance [4, 5, 6]. However, frequent and large magnitude violations can cause serious damage, and even crash the system [7]. For example, power capping systems deployed in hyperscale datacenters (e.g., Amazon, Google, Microsoft) smooth out spikes from occasional power overloading [8, 9], but cannot tolerate large-scale violations without causing significant degradation in application performance [10].

To ensure that the system executes optimally while meeting the safety constraint, existing resource managers use *samples from the resource space* (i.e., hardware parameters and their measured system behavior) of past executions to model system behavior (e.g., power and performance) as a function of hardware resource usage. The model is then used to dynamically adjust resources usage such that safety is maintained and application performance is optimized [6, 11, 12, 13, 14]. However, samples collected from the resource space may not be generalizable across different applications and inputs. This means that safe and high-performing hardware resource allocation for one execution can be unsafe and low-performing in another [15, 16]. Specifically, when a new application or input leads to significantly different system behavior, models learned using past executions cannot provide safety guarantees and optimal performance.

Execution space. To address this limitation, I introduce *execution space*, which is the cross product of hardware resources, input features, and applications. I define a *configuration* as a tuple of hardware resources, input features (e.g., data size), and application. To ensure that the system executes optimally while meeting the safety constraint, resource managers need to *explore* (i.e., evaluate a configuration that the system has not seen before) the execution space, rather than the resource space. Exploring the execution space is the process of evaluating a previously unseen configuration from the execution space and it allows us to learn models of system behavior as a function of both the hardware parameters, the current application, and input.

1.1 Proposed Approach: SCOPE

To dynamically and safely allocate hardware resources from the execution space, this thesis presents SCOPE [1], a resource manager that leverages a novel safe exploration framework. *Safe exploration* is a family of sequential decision-making techniques that optimize an objective while minimizing safety constraint violations [17, 18, 19]. Unlike static configuration that uses the same hardware resources throughout application execution, SCOPE dynamically reallocates hardware resources to optimize system performance and minimize safety constraint violations while responding to dynamic runtime changes. SCOPE operates iteratively, with each iteration (i.e., reallocation) having three phases: *monitoring, safe space construction, and objective optimization*.

- In the monitoring phase, SCOPE samples the execution space; i.e., it measures the system behavior for the current application, input, and hardware resource allocation. Specifically, it records both the objective and safety data, and checks whether the safety constraint has been violated or not.

[1] **S**afe **C**onfiguration **O**ptimization for **P**erformance and **E**fficiency

- Safe space construction is the process of predicting a *safe set* (i.e., a set of configurations that will not violate the safety constraint) with high coverage (i.e., a high number of safe configurations in the predicted safe set). Furthermore, if an unsafe configuration is included in the safe set, its violation magnitude should be small. To construct a safe set with these properties, SCOPE introduces a locality preserving operator based on the *locality preserving criterion* [20] (i.e., if two configurations are close in distance, their system behaviors are likely close as well). This operator constrains SCOPE to explore only configurations within a certain distance of the most recently executed safe configuration. Because any new configurations are close to known safe configurations, exploring these new configurations will rarely violate the safety constraint and have small magnitude violations if it does.
- In the objective optimization phase, SCOPE reallocates hardware resources with the best predicted performance from the newly constructed safe set.

1.2 Results and Contributions

This thesis evaluates SCOPE's ability to minimize latency (the objective) while meeting a power (the safety) constraint for a variety of Apache Spark applications [21]. For each input and application, SCOPE dynamically configures hardware resources (e.g., CPU frequency, uncore frequency, number of sockets, number of cores per socket, and whether hyperthreads are enabled). Compared to prior approaches that minimize power constraint violations, evaluation results show that on average:

- Compared to Intel's RAPL [22], SCOPE decreases the violation rate by 54.1× and violation magnitude by 1.04×. These violation reductions occur because SCOPE can reach even lower power caps than RAPL. Since RAPL does not optimize latency, SCOPE is able to decrease latency by 9.5×.
- Compared to an existing state-of-the-art safe exploration approach from domains outside of computer systems, SCOPE decreases latency by 1.11× across all evaluated applications while decreasing the violation rate and magnitude by 11.93× and 1.40× respectively. SCOPE achieves these results because it accounts for the unique features of com-

puter systems by continually monitoring the system; prior work assumes that samples taken early in execution accurately capture behavior over the system lifetime.

- With its locality preserving operator, SCOPE's predicted safe set has 1.96× higher coverage (i.e., a high number of safe configurations in the predicted safe set) and 1.35× lower violation magnitude than SCOPE-NO, a version of SCOPE that does not use the operator (section 6.3). This means that even in the rare case where an unsafe configuration is selected from SCOPE's predicted safe set, it will likely have lower violation magnitude than that of SCOPE-NO's.

The contributions of this thesis are as follows:

- Expanding the exploration space from resource space to execution space, which captures that system behavior is function of hardware resources, application, and input.
- Presenting SCOPE, a resource manager that leverages the safe exploration framework to optimize the objective while minimizing the safety constraint violations.
- Introducing the locality preserving operator to construct the safe set with high coverage.

Safe exploration is an important, emerging frontier of machine learning with a wealth of applications in safety-critical systems. To the best of my knowledge, SCOPE is the first demonstration of safe exploration in execution space for computer systems optimization. SCOPE outperforms existing safe exploration techniques from other domains by developing a locality preserving operator and requiring fewer assumptions about application behavior than other safe exploration techniques. This thesis offers a foundation on which the computer systems community can build new optimization tools that aid in exploration (to improve system performance) while preserving safety.

Chapter 2

Related Work

The key insight of this thesis is a methodology for using machine learning to perform optimal resource management while meeting safety constraints in execution space; i.e., without accounting for application and input. This section focuses on related work in machine learning for computer systems optimization (section 2.1, section 2.2). Rather than learning the relationships between all three components (i.e., hardware resources, input, and application) from the execution space, prior work learns from the resource space by assuming that samples of the past executions from the resource space can accurately predict the system behaviors of future executions from potentially different applications or inputs. I also explain the difference between existing safe exploration techniques (from domains other than computer systems) and SCOPE (section 2.3).

2.1 Machine Learning for Unconstrained Optimization

Machine learning techniques have been increasingly applied to solve computer systems optimization problems by modeling complex, nonlinear relationships between system resource usage and quantifiable behavior [23, 24]. Much prior work focuses on unconstrained optimization problems with no safety constraint—i.e., optimizing a single objective such as latency [25], throughput [9], power [26], and energy consumption [4]. These works share a common methodology of building machine learning models using training data collected by sampling the system resource space: allocating different system resources and measuring behaviors. Specifically, there are two types of system sampling, random and intelligent, and each leads to different types of machine learning approaches. Random sampling typically needs a large amount of samples to build a highly accurate model, but it is free of biases that might arise from intelligent sampling [27, 26, 28, 29, 30, 31, 32, 1, 33, 34, 35, 36, 37, 38, 39]. For example, Lee et al. [40] build a regression model on simulated data to predict multiprocessor performance. Paragon [41] and Quasar [42] apply collaborative filtering to predict QoS performance of workloads in datacenters. However, the large sampling effort required for random sampling is often prohibitive due to is high computational cost and inefficiency [16].

Unlike random sampling, intelligent sampling significantly reduces the sampling effort required to achieve good systems outcomes [11, 43, 44, 45, 46]. A representative family of intelligent sampling techniques is Bayesian optimization, which iteratively predicts and samples data points that contribute the most information to the learning model [47]. CherryPick [48] and CLITE [49] use Bayesian optimization to schedule workloads in datacenters. GIL [15] and Bliss [50] use Bayesian optimization to optimize system performance. HyperMapper [51] and BOCA [52] use Bayesian optimization to tune compilers. These works reduce the number of samples required to perform optimization, but do not consider any safety constraints. This motivates the need for a learning-based resource manager that both (1) works with reduced number of samples and (2) respects safety constraints.

2.2 Machine Learning for Safe Optimization

Safe optimization problems in computer systems find the optimal point within a tradeoff space (e.g., performance versus power)—i.e., optimizing a performance objective under some safety constraint [53, 54, 13, 1, 14]. For example, Li and Martinez [53] collect samples to optimize power under a performance constraint. Dubach et al. [54] collect samples to build a dynamic control system that optimizes energy and performance efficiency. LEO [12] and CALOREE [6] develop hierarchical Bayesian models to meet latency constraints and minimize energy.

These works provide safety under the assumption that the samples they collect from the resource space of past executions can accurately capture the system behaviors that will be seen during future executions. However, this assumption can be violated when a new application or new input causes significantly different system behaviors than those in the collected samples. The emergence of such unsampled behavior would render the whole system unsafe. As such, there is a need for approaches that can explore unseen configurations in the execution space, rather than the resource space.

2.3 Safe Exploration in Other Problem Domains

Recent years have witnessed safe exploration applied to safety-critical domains such as autonomous driving [55], healthcare [18], and robotics [56]. Safe exploration is a family of sequential decision-making techniques that optimize an objective while minimizing safety constraint violations [57, 17, 18, 19, 58, 59, 60, 61]. To satisfy some safety property with a high probability, these techniques either require extra supervision or knowledge accumulated before exploration. They achieve probabilistic (i.e., not deterministic) safety guarantees based on the assumption that the changes of safety measurements are continuous and bounded (i.e., Lipschitz continuity). In computer systems, however, this assumption does not hold. A typical example is power, where the power usage can change dramatically every second [62]. Different from prior work, my proposed approach, SCOPE, is a new safe exploration framework that introduces a locality preserving operator to eliminate the need for such assumptions. Evaluation results show that SCOPE outperforms the existing state-of-the-art safe exploration technique (e.g., StageOPT [18]) in both performance and safety by tailoring our approach to the unique properties of computer systems.

Chapter 3

Motivational Examples

I use two examples to demonstrate that to explore new configurations safely, optimization must operate in execution space rather than resource space because safe samples from the resource space of past executions do not generalize to new applications and new inputs. To demonstrate a lack of generalization across applications, I run two different applications and show that the safe samples from the resource space for one application are no longer safe for another. To demonstrate a lack of generalization across inputs, I run one application with two different inputs and show that the safe samples from the resource space for one are not safe for the other.

I consider two Apache Spark applications from HiBench [63] and run them on the Chameleon configurable cloud computing platform [64] (details in section 5.1). I collect samples by profiling the application at all possible assignments of hardware parameters (see Table 5.1) and recording their latency and power data. The goal is to minimize latency while meeting a power constraint.

Safe samples from the resource space do not generalize across applications. I show that safe samples from the resource space do not generalize across the applications `als` and `nweight`. I construct a safe set for `als` by randomly selecting a list of hardware allocations that do not violate the safety constraint based on all samples collected for `als`. Then, I dynamically configure `als` (at 20s intervals) using this safe set. I then use the same safe set to dynamically configure `nweight` (again, at 20s intervals).

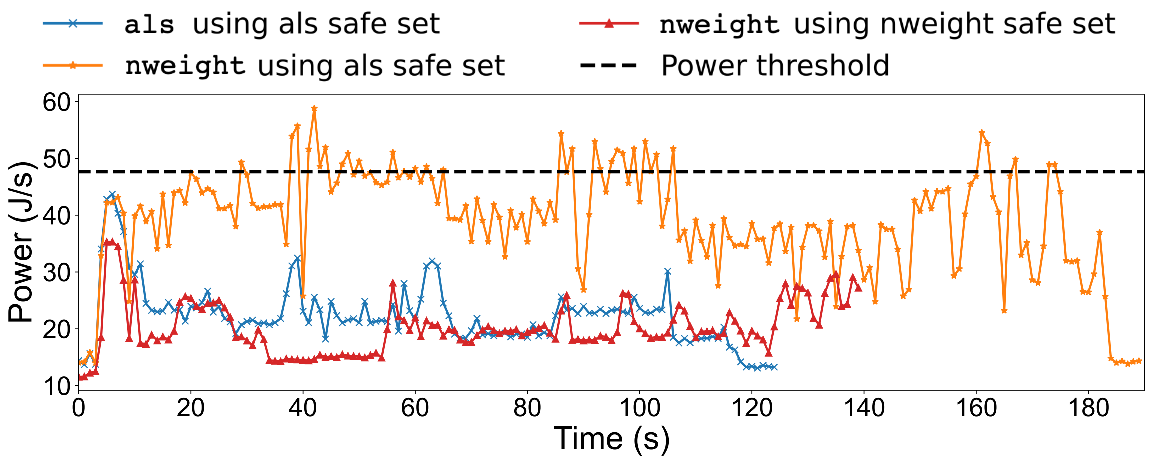

Figure 3-1: Results of power and latency for executing `als` and `nweight`.

Figure 3-1 shows the resulting power as a function of execution time, where the x-axis is the execution time, and the y-axis is the power. The blue line represents `als` using its own safe set, the orange line represents running `nweight` using `als`'s safe set, the red line represents running `nweight` using safe hardware allocations constructed especially for `nweight` (again by profiling all possible assignments of hardware parameters). The black dotted horizontal line is the power limit which represents the safety constraint in this example. The results show that `als` finishes safely in 125s with 0 power violations, while `nweight` using `als`'s safe hardware allocations needs 190s despite 32 power violations.

The high number of violations and the significant difference in latency, demonstrates that the safe samples from the resource space do not generalize across different applications. In fact, `nweight`—with a properly constructed safe set—finishes in 142s with 0 violations, indicating that it is possible to improve `nweight`'s latency and safety, however, exploration operating only within the resource space fails to do so.

Safe samples from the resource space do not generalize across different inputs for the same application. I show that safe samples from the resource space do not generalize across different inputs for the same application using `als`. I create two different inputs with the same sizes (so the execution behavior variations are due to properties of the data rather than data size): `Input A` is the input we will sample to build a safe set and `Input B` is the target input to be optimized. I construct a safe set for `Input A` by randomly selecting a list

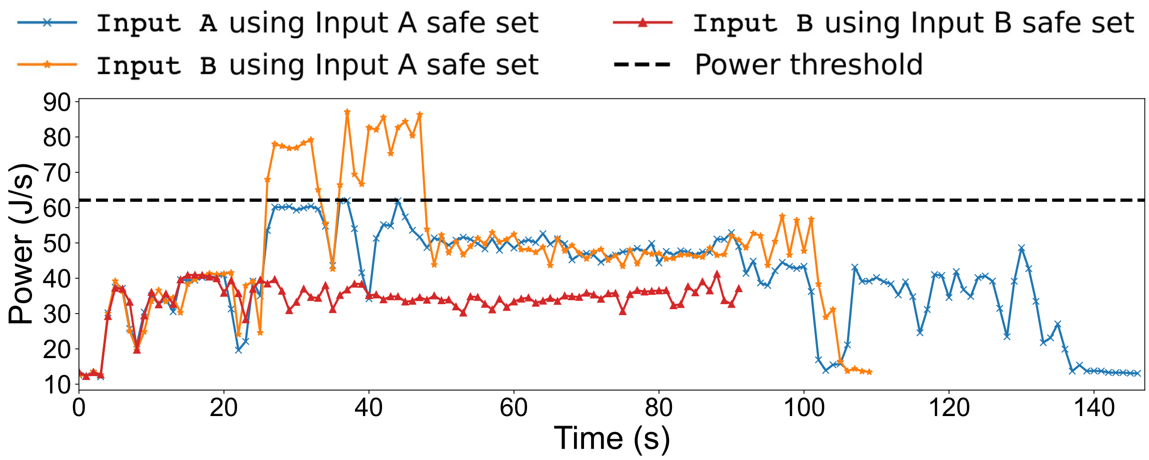

Figure 3-2: Results of power and latency for executing `als` using two different sets of inputs with the same size.

of hardware allocations that do not violate constraints based on all samples we collected for `Input A`. Then, I dynamically configure for `Input A` (at 20s intervals) using this safe set, and then configure using the same safe set for `Input B` (again, at 20s intervals).

Figure 3-2 shows the the resulting power as a function of execution time, where the x-axis is the execution time, and y-axis is the power. The blue line represents running `Input A` using its own safe set, the orange line represents running `Input B` using `Input A`'s safe set, the red line represents running `Input B` with its own safe set found by evaluating `Input B` in all possible assignments of hardware parameters. The black dotted horizontal line is the power threshold, or safety constraint in this example. The results show that `Input A` finishes in 147s with 0 violation. Running `Input B` with `Input A`'s safe set, however, finishes in 110s with 20 violations, mainly between 26s and 46s. `Input B` is mostly under the power threshold, but its dynamic behavior causes it to violate safety constraint, while `Input A` does not.

The high number of violations when running `Input B` demonstrates that safe samples from the resource space for `Input A` are no longer safe with different inputs, even when the inputs are the same size. In fact, `Input B` can finish in 91s with 0 violations with an appropriate safe set, which indicates that the safe samples from the resource space for `Input A` do not generalize to `Input B`.

These examples illustrate that safe samples from the resource space of past executions fail to generalize to new applications and new inputs. As an alternative to generalizing safe samples to new applications or inputs, the next section describes SCOPE, the proposed solution to optimize objectives and minimize safety constraint violations in the execution space, rather than resource space.

Chapter 4

SCOPE Design

SCOPE is a resource manager that dynamically and safely explores in the execution space. In other words, SCOPE makes no assumptions about how past executions, with different applications or inputs, affect the current system behavior. SCOPE iteratively reallocates hardware resources, with each iteration having three phases: monitoring, safe space construction, and objective optimization. Figure 4-1 illustrates SCOPE's workflow. In the first phase of each time interval, SCOPE continually measures safety and objective metrics for the current application, input, and hardware resource allocation (A), and checks whether the safety constraint has been violated or not (1); if a safety measurement violates the safety constraint, SCOPE moves to the next phase immediately, otherwise it waits for a fixed time interval to expire. Then, SCOPE goes to the phase of safe space construction to predict a safe set with high coverage. Within this phase, SCOPE first builds a safety model using the measured configurations and safety data (B), and then constructs a safe set based on the safety model (C). Then, SCOPE's objective optimization phase reconfigures the system. Within this phase, SCOPE first builds an objective model using the measured configurations and objective data (D), and then picks a predicted high-performing configuration from the safe set and puts the system into that configuration.

The remainder of this section first sets up the core concepts (section 4.1), and then describes SCOPE in detail (section 4.2, section 4.3, section 4.4, and section 4.5).

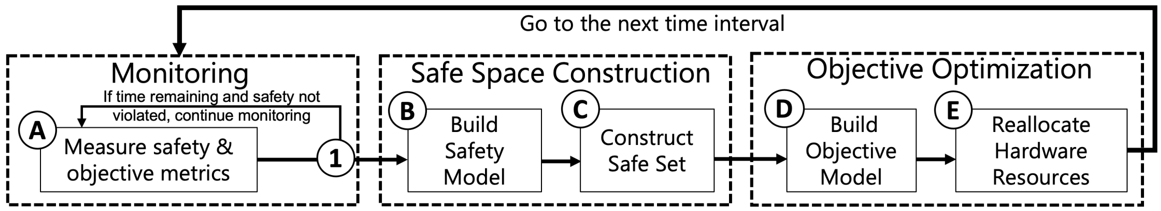

Figure 4-1: The workflow of SCOPE at each time interval while application is executing.

4.1 Background and Definitions

SCOPE's input includes an application, objective metric to optimize, safety constraint, a starting safe configuration, number of measurements for each time interval, and a list of configurations over which to optimize. A starting safe configuration is needed to prevent SCOPE from violating the safety constraint during the first iteration; users can conservatively choose this configuration. For example, if the safety metric is power, users could start in a configuration with minimal hardware resources. If latency is the safety metric, then users could start with a configuration that makes all resources available. SCOPE's exploration will safely move the system out of this conservative configuration to one that improves the objective metric.

Application. A program that runs on a computer system using hardware resources.

Configuration. A configuration $\mathbf{x}_i \in D$ is a d-dimensional vector that includes d parameters: $\mathbf{x}_i = [x_{i1}, x_{i2}, \ldots, x_{id}]$, where i is the i-th configuration, and x_{ij} is the respective value for the j-th parameter, $j \in [d]$.

Safety constraint. The safety constraint is a threshold for any particular metric that the system does not want to violate during application execution. Some common metrics that can be used for safety constraints include energy [65, 13], power [66], and latency [12, 6, 14]. In this thesis, I use P to denote the safety constraint and y_i to denote the safety measurement for the i-th configuration.

Safe configuration. The safe configuration is a configuration that does not violate the safety constraint, i.e., $y_i < P$. Similarly, an unsafe configuration is a configuration that violates the safety constraint, i.e., $y_i \geq P$.

Optimization objective. The optimization objective is the metric that SCOPE minimizes or maximizes. Some common optimization objectives include energy [12, 6, 14], power [66], and latency [65, 13]. In this thesis, I use z_i to denote the optimization objective measurement for the i-th configuration.

Time interval. The time interval is a short period of time during application execution. In this thesis, I use T_i to denote i-th time interval when the application runs configuration $\mathbf{x}_i$. During each time interval, SCOPE will execute all three of its phases: monitoring, safe set construction, and objective optimization.

Measurement interval. The measurement interval is the amount of time over which SCOPE gets a pair of safety and objective measurements.

4.2 Monitoring

At i-th time interval when the application runs at configuration $\mathbf{x}_i$, SCOPE continuously gets safety y_i and objective z_i at each measurement interval for a maximum of N times, where N is an input parameter of SCOPE (A). To react to violations as quickly as possible, if a safety measurement violates the safety constraint, SCOPE stops monitoring and moves to the next phase immediately instead of finishing N measurements (1). The measurement data collected over the time intervals are the training data for the next phases of SCOPE; i.e., until the i-th time interval, $X_{\text{train}} = (\mathbf{x}_j)_{j=0}^{i}$, $Y_{\text{train}} = (y_j)_{j=0}^{i}$, $Z_{\text{train}} = (z_j)_{j=0}^{i}$.

4.3 Safe Space Construction

After getting the training data from measurement, SCOPE goes to safe space construction, the process of predicting a safe set with high coverage—i.e., a high number of safe configurations in the predicted safe set—and if an unsafe configuration is included, its violation magnitude should be small. To construct a safe set with these properties, SCOPE introduces a *locality preserving operator* based on the *locality preserving criterion* [20] (i.e., if two configurations are close in distance, their system behavior is also likely to be close).

The translation of this criterion to safe space construction is as follows:

- If two configurations are close in distance, their corresponding safety and objective measurements are likely close too.
- If configuration A is safe and configuration B is unsafe but close to configuration A in distance, the safety violation of configuration B is very likely to be small.

This criterion matches my empirical observation that a smaller magnitude configuration change (e.g., changing 2 cores to 4 cores) leads to smaller system behavior changes than a larger change (e.g., changing 2 cores to 12 cores). The locality preserving operator constrains SCOPE to explore a subset of configurations in the safe set that are within a neighborhood (i.e., close area within some distance) of the most recently used safe configuration so that SCOPE's exploration will rarely violate the safety constraint and have small magnitude violations if it does. Formally, X_c is the candidate set that is constructed with the locality preserving operator:

$$X_c = \{\mathbf{x} \in D \setminus X_{\text{train}} \mid \|\mathbf{x} - \mathbf{x}^s\| \leq \gamma\}, \tag{4.1}$$

where $\mathbf{x}^s$ is the most recent safe configuration that SCOPE has been used, and $\gamma \geq 0$ is the operator parameter. The operator parameter γ controls the distance that SCOPE explores. If $\gamma = 0$, the only configuration that SCOPE can explore is the starting safe configuration. If $\gamma > 0$, SCOPE can explore configurations that are within distance γ from $\mathbf{x}^s$. The higher γ is, the larger configuration space SCOPE can explore, and thus the more likely the unsafe configurations can be included in the safe set. When γ is large enough, it is equivalent to exploring without constraints.

Upon obtaining the candidate set X_c, SCOPE constructs the safe set as follows. At i-th time interval T_i, SCOPE trains the safety model f_y using the configurations and safety measurements collected so far $(X_{\text{train}}, Y_{\text{train}}) = (\mathbf{x}_j, y_j)_{j=0}^{i}$ (B). SCOPE uses this model to predict the safety values of the configurations in the candidate set X_c, and include configurations that are predicted to meet the safety constraint in the safe set (C):

$$X_s = \{\mathbf{x} \in X_c | f_y(\mathbf{x}) < P\}, \tag{4.2}$$

where X_s is the safe set, and P is the safety constraint. The design of safe space construction is compatible with any type of learning models such as Gaussian process regression [48], random forest [51], linear model [15], and neural networks [11]. SCOPE uses Gaussian process regression since it performs the best in practice (section 6.6).

4.4 Objective Optimization

The objective optimization phase reallocates hardware resources with a high-performing safe configuration. At i-th time interval, SCOPE trains the objective model f_z using the configurations and objective measurements collected so far $(X_{\text{train}}, Z_{\text{train}}) = (\mathbf{x}_j, z_j)_{j=0}^{i}$ (D). SCOPE uses this model to predict the objective values of the configurations from the safe set. Empirically, we find that the safe set could be empty when the operator parameter γ is too small, since there is not much neighborhood for SCOPE to explore. To address the possible situation like this, SCOPE does the following.

- If the safe set is empty, SCOPE chooses the configuration in the candidate set X_c that has the best predicted safety. In this way, SCOPE is able to explore the configuration space while avoiding safety violations.
- If the safe set is not empty, SCOPE picks the configuration in the safe set $X_s = \{\mathbf{x} \in X_r | f_y(\mathbf{x}) < P\}$ that has the best predicted objective.

After SCOPE picks the new configuration and reallocates hardware sources based on this configuration (E), it goes to the next time interval until the execution ends.

4.5 SCOPE Algorithm Summary

SCOPE is a general system design compatible with any type of learning models, safety constraints, and optimization objectives. To demonstrate its effectiveness in solving systems problems, this thesis uses SCOPE to minimize latency while constraining the power usage to be below a fixed threshold, where the safety metric is power, and the optimization objective is work done so far (e.g., total instruction count). The input includes a starting safe configuration $\mathbf{x}_0$, safety constraint P, number of measurements N, and operator param-

Algorithm 1 The SCOPE Resource Manager

Require: $\mathbf{x}_0$ ▷ Starting safe configuration.
Require: P ▷ Safety constraint threshold.
Require: γ ▷ Operator parameter.
Require: N ▷ Number of measurements.
1: $X_{\text{train}} = \{\}$ ▷ Set of sampled configurations.
2: $Y_{\text{train}} = \{\}$ ▷ Set for safety measurements.
3: $Z_{\text{train}} = \{\}$ ▷ Set for objective measurements.
4: $i = 0$
5: $\mathbf{x}^s \leftarrow \mathbf{x}_0$ ▷ Assign current safe configuration.
6: Run application at $\mathbf{x}_0$.
7: **while** application running **do**
8:     **for** $t = 1,...N$ **do**
9:         Get safety y_i and objective z_i.
10:         **if** $y_i > P$ **then**
11:             Stop monitoring.
12:     Update training set $(X_{\text{train}}, Y_{\text{train}}, Z_{\text{train}})$ with $(\mathbf{x}_i, y_i, z_i)$.
13:     Train safety model f_y using X_{train} and Y_{train}.
14:     **if** $y_i < P$ **then**
15:         $\mathbf{x}^s \leftarrow \mathbf{x}_i$ ▷ Update current safe configuration.
16:     $X_c = \{\mathbf{x} \in D \setminus X_{\text{train}} | \, ||\mathbf{x} - \mathbf{x}^s|| \leq \gamma\}$ ▷ Update candidate configuration set.
17:     $X_s = \{\mathbf{x} \in X_c | f_y(\mathbf{x}) < P\}$ ▷ Construct safe set.
18:     Train objective model f_z using X_{train} and Z_{train}.
19:     **if** $|X_s| == 0$ **then**
20:         $\mathbf{x}_{i+1} \leftarrow \arg\min_{\mathbf{x} \in X_c} f_y(\mathbf{x})$ ▷ If safe set is empty, pick configuration with best predicted safety.
21:     **else**
22:         $\mathbf{x}_{i+1} \leftarrow \arg\max_{\mathbf{x} \in X_s} f_z(\mathbf{x})$ ▷ If safe set is not empty, pick configuration with best predicted objective.
23:     Reallocate hardware resources with $\mathbf{x}_{i+1}$.
24:     $i \leftarrow i + 1$

eter γ. The starting safe configuration can be set by the user since it is not desirable that the application violates the safety constraint in the beginning. The safety constraint depends on the optimization goal and is set by the user. The operator parameter γ is set by the user to control the exploration space, where more insights can be found in section 6.4.

Algorithm 1 summarizes the procedure. In line 6, SCOPE starts executing the application, and records the current time T_0. While the application is executing, SCOPE does the following steps iteratively. For each measurement interval, SCOPE gets the safety and objective (line 9). If the safety measurement violates the safety constraint (line 10), SCOPE stops measuring and moves to the next phase (line 11). Otherwise, SCOPE continues measuring. In line 12, SCOPE updates the training set by adding the new measured data. In line 13, SCOPE trains the safety model using training configurations and safety data. In line 15, SCOPE updates the safe configuration that has been most recently executed. In line 16, SCOPE updates the candidate set by applying the locality preserving operator. In line 17, SCOPE constructs the safe set from the candidate set based on the predictions using the safety model. In line 18, SCOPE trains the objective model using training configurations and objective data. Then, SCOPE will conduct different steps based on the cardinality of the safe set. In lines 19 and 20, if the safe set is empty, SCOPE will pick the configuration with the best predicted safety in the candidate set. Otherwise, in line 22, SCOPE will pick the configuration with the best predicted objective in the safe set. In line 23, SCOPE reallocates hardware resources based on the newly pickled configuration. I implement SCOPE in Python with libraries including numpy [67], pandas [68], and scikit-learn [69]. The code is released in `https://github.com/kim1031/scope`.

Chapter 5

Experimental Setup

5.1 Hardware System

This thesis experiments on the Chameleon configurable cloud computing platforms [64], where each experiment runs on a master node and four worker nodes. Each node is a dual-socket system running Ubuntu 18.04 (GNU/Linux 5.4) with 2 Intel(R) Xeon(R) Gold 6126 processors, 192 GB of RAM, hyperthreads and TurboBoost. Each socket has 12 cores/24 hyperthreads and a 20 MB last-level cache. I tune the hardware parameters in Table 5.1, as they have been shown to influence both latency and power tradeoffs and are important to tune to optimally meet a power cap [70]. In total, there are 1920 possible allocations of hardware resources to be explored dynamically while minimizing power cap violations.

Table 5.1: Hardware parameters.

Parameter	**Range**
CPU frequency (GHz)	1.0–3.7
Uncore frequency (GHz)	1.0–2.4
Hyperthreading	on, off
Number of sockets	1, 2
Number of cores per socket	1–12

5.2 Software System

For the experiments, I use Apache Spark [21] with the default Spark configuration settings as the software system. I test 12 applications from HiBench's benchmark suite [63], which has been widely applied to configuration optimization evaluations [16, 71, 15, 72]. The applications cover various domains including microbenchmarks, machine learning, web-search, and graph analytics (Table 5.2).

Table 5.2: HiBench applications.

Application	**Data size**	**Application**	**Data size**
als	0.7 GB	bayes	1 GB
gbt	0.1 GB	kmeans	2.1 GB
linear	36 GB	lr	2 GB
nweight	0.1 GB	pagerank	0.2 GB
pca	0.1 GB	rf	1.6 GB
terasort	3.2 GB	wordcount	26 GB

5.3 Points of Comparison

This thesis compares various approaches including static configuration, approaches with extensive collected samples (Offline), approaches without collected samples (RAPL, BO, StageOPT, SCOPE-NO), and an approach with perfect knowledge (Oracle, which I create with brute force search and is, of course, unrealizable in practice).

- **Static**: run application at the starting safe configuration throughout the execution.
- **Offline**: before running the target application, randomly sample half of all possible hardware resource allocations using the target application and input and build safety and objective models using Gaussian process regression. These models are not updated during application execution. Offline reconfigures using the predictions from these models at each time interval [12, 6].
- **RAPL**: Intel's Running Average Power Limit system that allows users to set a power limit and tunes processor behavior to respect that limit using dynamic voltage and frequency scaling technique [22]; RAPL only configures CPU and uncore frequency.

- **BO**: use Bayesian optimization to reconfigure at each time interval, with Bayesian Gaussian process regression being the learning model and expected improvement being the acquisition function [48, 51, 49, 52].
- **StageOPT**: use the StageOPT algorithm [18], a representative of safe exploration approaches from other domains that assume continuous and bounded changes in safety measurements to achieve probabilistic safety guarantee. StageOPT separates safe set construction and objective optimization into two stages: in the first few iterations, it focuses on constructing the safe set only; then, it switches to optimizing the objective within the safe space without updating its predicted safe set. The lack of update to the safe set can be very problematic in computer systems. As shown in the motivational example, the discrete nature of computer systems means that applications that look safe might quickly change to unsafe with no prior warning (see Figure 3-2). In contrast, SCOPE continually updates its safe set.
- **SCOPE-NO**: use the introduced safe exploration framework without the locality preserving operator to reconfigure at each time interval.
- **SCOPE**: use the introduced safe exploration framework with the locality preserving operator to reconfigure at each time interval. Both SCOPE and SCOPE-NO use Gaussian process regression as the learning model since it performs the best (section 6.6).
- **Oracle**: profile all latency and power data for the entire configuration space, and pick the fastest configuration that meets the power constraint.

5.4 Evaluation Methodology

I start executing the application at the same safe configuration for each approach in section 5.3, and then reconfigure it dynamically at each time interval. For all approaches that reconfigure dynamically, I run the sweep over different time intervals and pick the best time interval for each approach (section 6.5). For Offline, I also sweep over different amounts of samples and choose the best amount for sampling (section 6.8). During each time interval, I use the maximum power for the safety and average of the work done (i.e., total instruction counts) for the objective.

This thesis evaluates on a wide range of power constraints that are set as [40, 50, 60, 70, 80]-th percentiles of the power distributions that are achievable across all configurations. This is a reasonable range in that a small constraint value (e.g., [10, 20, 30]-th percentiles) leaves little room for tuning and a large constraint value (e.g., [90]-th percentile) is often too relaxed to constrain the power.

To make a comprehensive comparison, I identify 5 starting configurations as fast and 5 as slow. For fast configurations, the latency is below p35 of the latency distribution over all configurations that meet the power constraint. For slow ones, the latency is above p65 of the latency distribution that meet the power constraint. The reported results are averaged over different constraints and 10 different starting safe configurations.

For SCOPE, I choose the operator parameter $\gamma = 1$ for all applications based on the best tradeoffs between the violation rates and speedups from sensitivity analysis in section 6.4. I use 1 second as the measurement interval.

It should be noted that SCOPE runs on the same hardware whose power it is controlling. Thus, SCOPE must account and compensate for its own power overhead. All results include the power and latency overhead of running SCOPE.

5.5 Evaluation Metrics

For latency evaluation, I measure the latency l_{confg} obtained by each approach, and compute its speedup compared to the latency l_{static} of Static:

$$\text{speedup} = \frac{l_{\text{static}}}{l_{\text{confg}}}. \tag{5.1}$$

For power evaluation, I record the power over 1 second interval. To measure how much the system violates a power constraint, I note the times that power exceeds the power threshold n_{violate}, and then divide it by the total number of measurements n_{total}:

$$\text{violation rate} = \frac{n_{\text{violate}}}{n_{\text{total}}} \times 100\%. \tag{5.2}$$

I also record the average of power usage that exceeds the power threshold p throughout application execution, and then divide it by the power threshold P:

$$\text{violation magnitude} = \begin{cases} 0 & \text{if } p \leq P \\ \frac{p}{P} & \text{if } p > P \end{cases} \tag{5.3}$$

For safe set, the coverage of the safe set is defined as follows:

$$\text{coverage} = \frac{m_{\text{safe}}}{m_{\text{total}}} \times 100\%, \tag{5.4}$$

where m_{safe} is the number of true safe configurations in the predicted safe set, and m_{total} is the number of all configurations in the predicted safe set.

Chapter 6

Experimental Evaluation

This thesis evaluates the following research questions (RQs):

- **RQ1:** Does SCOPE reduce power violations? SCOPE reduces violation rates by 3.62–54.1× (Figure 6-1) and violation magnitudes by 1.04–1.49× (Figure 6-2) on average, compared to other baselines.
- **RQ2:** Does SCOPE improve application latency? SCOPE improves application latency by 1.07–9.5× compared to other baselines (Figure 6-6).
- **RQ3:** Does locality preservation produce better safe sets? With the locality preserving operator, SCOPE's safe set has 1.96× higher coverage and 1.35× lower violation magnitude than SCOPE-NO that does not use the operator (Table 6.1), which means that even in the rare case where an unsafe configuration is selected from SCOPE's safe set, it will likely have lower violation magnitude than that of SCOPE-NO.
- **RQ4:** How sensitive is SCOPE to γ? The operator parameter γ affects both speedup and violation rate, and all applications share a common trend of the tradeoffs between speedup and violation rate (Figure 6-8).
- **RQ5:** How sensitive is SCOPE to time intervals? SCOPE is the most robust to the time interval between reallocations compared to other baselines due to its ability to update models at all iterations and locality preserving operator (Figure 6-9).
- **RQ6:** How do different types of models perform? SCOPE can be used with any types of learning models, and I chose Gaussian process regression for all evaluations due to its lowest violation rates and comparably low latency (Figure 6-10).

- **RQ7:** What are the overheads? SCOPE achieves low overhead of 0.05s per sample on average, which is negligible given the fact that I include the overheads in all experiments and SCOPE has the best latency improvement (Figure 6-11).

6.1 RQ1: Does SCOPE reduce power violations?

I use violation rate and magnitude to evaluate how well SCOPE reduces power violations. Figures 6-1 and 6-2 summarize average violation rates and magnitudes for different approaches, where the x-axis is the application, the y-axis is the violation rate or magnitude, the last column Mean is the arithmetic mean over all applications for Figure 6-1 and arithmetic mean over all applications with non-zero violation magnitude for Figure 6-2, where I put the number on each bar to quantify the mean results. To visualize better, I cap the violation rate at 5% in Figure 6-1 and put numbers on the bars that are capped. Detailed results of violation rates and magnitudes for each approach under different power constraints (p40-80) are shown in Figures 6-3 and 6-4. Static and Oracle were omitted since they do not violate at all; Static runs the same starting safe configuration throughout the execution, and Oracle runs the fastest configuration that is safe based on the exhaustive search. Figure 6-5 shows power results at every time point for different approaches, where x-axis is the running time, and y-axis is the power. Overall, SCOPE has the lowest violation rate and magnitude at almost all power constraints, showing that SCOPE not only makes rare and small disruptions that disturb the system the least but also has a dominating advantage over other approaches no matter what safety constraint is used.

Violation rate. SCOPE is 3.62× better than Offline, 54.1× better than RAPL, 45.88× better than BO, and 11.93× better than StageOPT on average.

Violation magnitude. SCOPE is 1.05× better than Offline, 1.04× better than RAPL, 1.49× better than BO, 1.40× better than StageOPT on average.

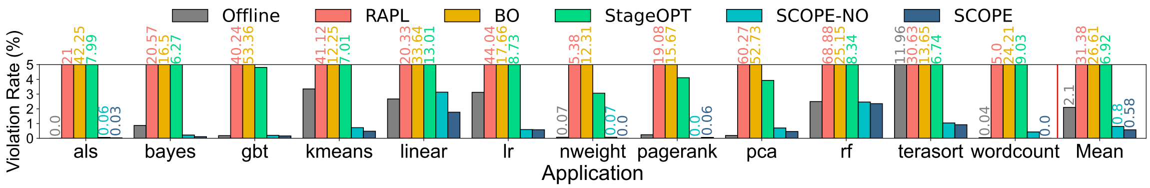

Figure 6-1: Violation rates of each approach averaged over all power constraints for each application. Lower is better.

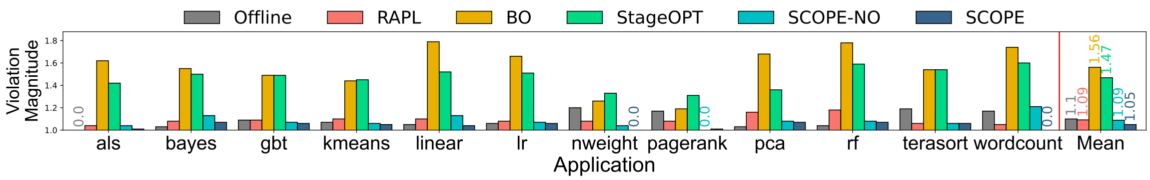

Figure 6-2: Violation magnitudes of each approach averaged over all power constraints for each application. Lower is better.

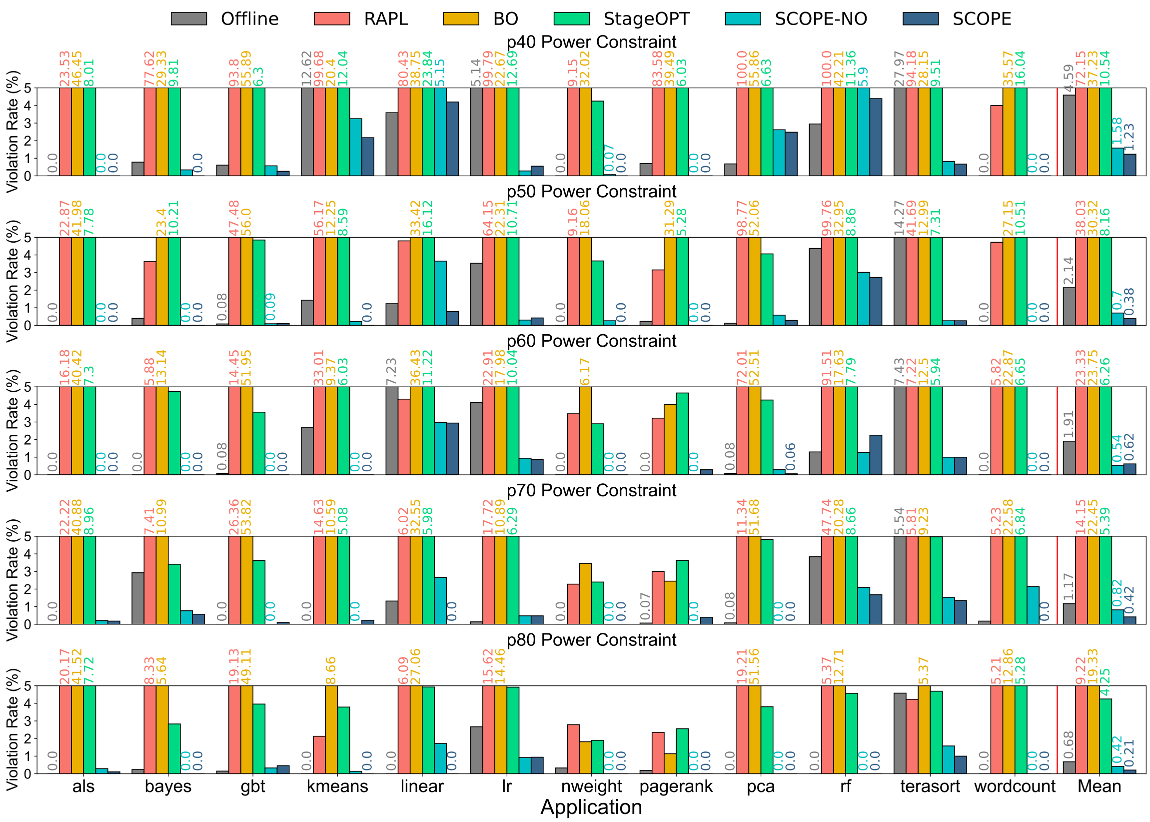

Figure 6-3: Violation rates averaged over all random starting configurations for each power constraint and application. Lower is better.

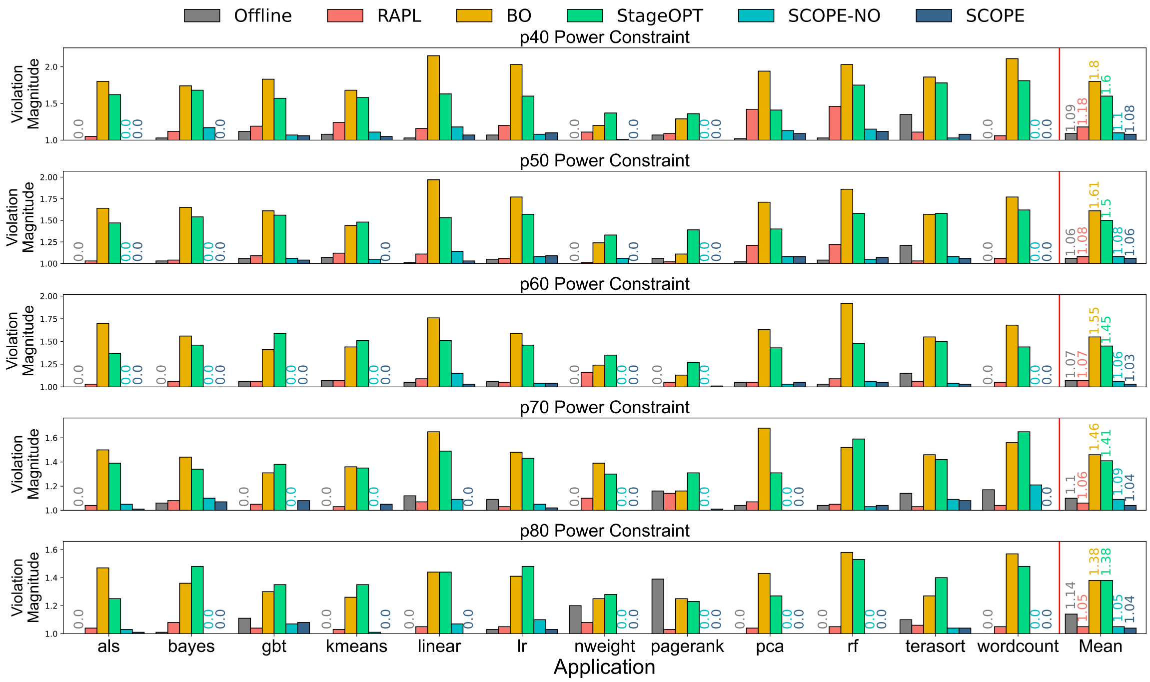

Figure 6-4: Violation magnitudes averaged over all random starting configurations for each power constraint and application. Lower is better.

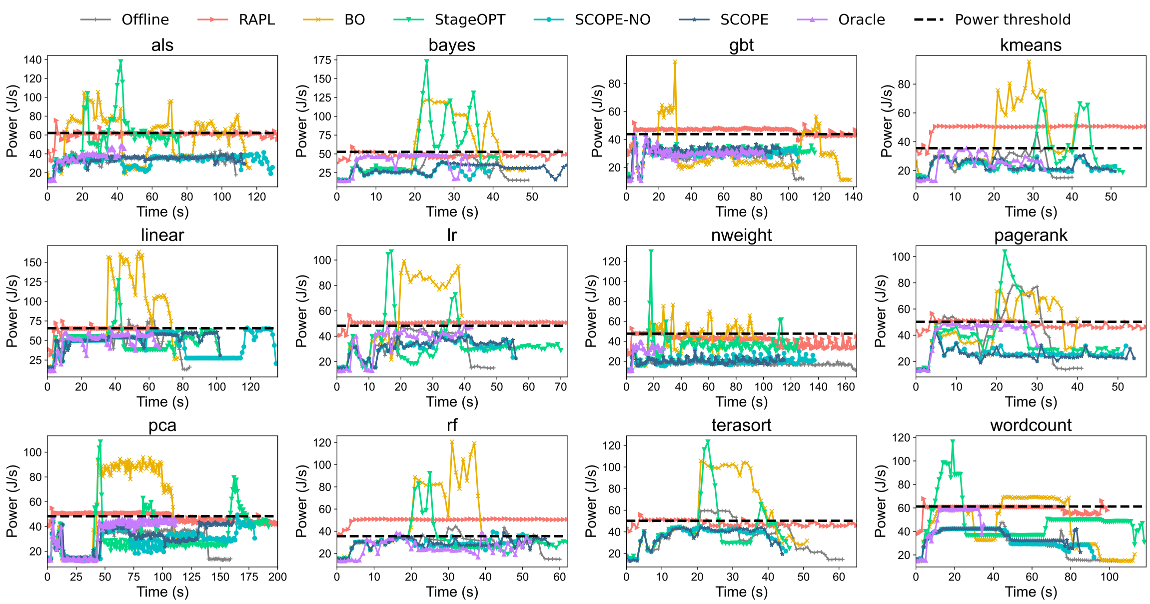

Figure 6-5: Power results at different time points during execution for each application.

More specifically, the results demonstrate the following:

- Offline underperforms SCOPE in all but 2 applications. Despite utilizing a large amount of samples collected a priori, the models Offline builds a priori do not generalize to the environment changes that occur during execution, while SCOPE dynamically updates its models to reconfigure. As a result, Offline has higher violation rate, which also demonstrates the difficulty of generalization of collected samples.
- RAPL has the highest violation rate despite the fact that it only optimizes for power. RAPL achieves low violation rate with higher power constraints (e.g. `nweight` and `terasort` in Figure 6-5), but the lower power constraints (i.e. [40,50]-th percentiles of the power distribution) are below the minimum power threshold that RAPL can meet and cause RAPL to violate constantly. This suggests that SCOPE performs much better than RAPL with a wider range of constraints.
- BO has the second highest violation rate and largest violation magnitudes since it optimizes for latency only and does not consider power constraints.
- StageOPT underperforms SCOPE because StageOPT achieves probabilistic safety guarantees by assuming continuity and boundedness of power measurements, which are not guaranteed to hold in computer systems. Thus StageOPT makes inaccurate predictions, which leads it to have higher violation rates and magnitudes.
- SCOPE has 1.38× lower violation rate and 1.04× lower violation magnitude than SCOPE-NO. The results suggest that the introduced operator is beneficial for reducing violations (more details in section 6.3).

6.2 RQ2: Does SCOPE improve application latency?

Figure 6-6 summarizes the speedups over Static for different approaches and Figure 6-7 details the speedups over Static for different approaches under different power constraints (p40-80), where the x-axis is the application, the y-axis is the speedup, and the last column Mean is the arithmetic mean over all applications, where I put the number on each bar to quantify the summarized results. I have omitted BO as it optimizes for latency only and thus has a very high violation rate and magnitude, which makes it an unfair comparison.

Oracle has the highest latency speedup since it has perfect prior knowledge to choose the fastest safe configuration. Overall, SCOPE is second to Oracle: 1.07× higher speedup than Offline, 1.11× higher speedup than StageOPT, and 9.5× higher speedup than the slowest baseline, RAPL. In particular, the results show the following:

- All approaches outperform Static except RAPL, which demonstrates the effectiveness of dynamic reconfiguration during execution for optimizing latency. RAPL is the exception because it manages power only and disregards latency. In addition, because RAPL only configures core frequency, it is unable to take advantage of more complex tradeoffs that can reduce power without harming latency as much (for example, reducing core usage for applications with low parallel speedup) [70].
- Offline, despite training on a large amount of samples collected prior to running the target application, has lower speedup than SCOPE. This is because Offline uses fixed models trained over early collected data that fail to adapt to changes in execution.
- StageOPT, despite achieving good results in other problem domains, underperforms SCOPE for computer systems optimization. It is because StageOPT only constructs the safe set in the first stage and does not optimize latency until the second stage, while SCOPE constructs the safe set and optimizes latency throughout application execution.
- SCOPE achieves 1.05× higher speedup than SCOPE-NO. This suggests that the locality preserving operator not only reduces safety violations, but improves latency. Detailed analyses can be found in section 6.3 and section 6.4.
- SCOPE is second to Oracle in all but the p50 power constraint, where Offline has slightly higher speedup. This suggests that SCOPE's dominating advantage is robust to the value of the safety constraint.

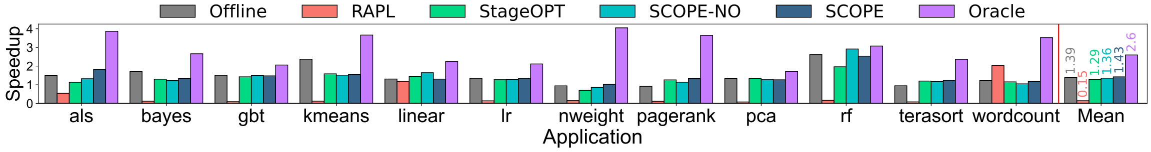

Figure 6-6: Speedup of each approach averaged over all power constraints for each application. Higher is better.

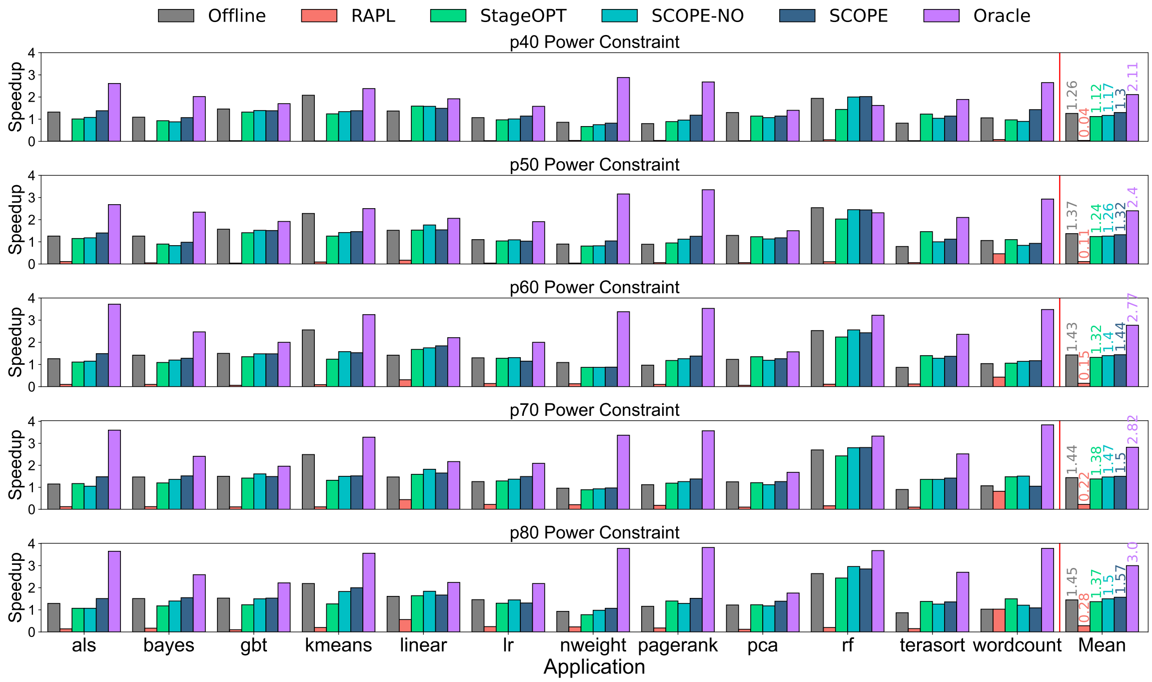

Figure 6-7: Speedup averaged over all random starting configurations for each power constraint and application. Higher is better.

6.3 RQ3: Does locality preservation produce better safe sets?

To better understand how the locality preserving operator improves the safe set for SCOPE over SCOPE-NO (the one without the operator), this section shows the percentage of configurations (POC) selected in the total configuration space, coverage, and the average violation magnitudes (AVM) of unsafe configurations in the safe set for SCOPE-NO and SCOPE in Table 6.1, where the numbers are averaged over different power thresholds and starting configurations. SCOPE's results are obtained when the operator parameter $\gamma = 1$ for all applications based on the best tradeoffs between violation rate and speedup (section 6.4). The results show the following:

- SCOPE has 4.9× POC smaller than SCOPE-NO, which indicates that the operator significantly reduces the number of configurations included in the safe set.
- Although SCOPE's safe set has significantly fewer configurations, its coverage is 1.96× higher than that of SCOPE-NO, showing that the operator greatly improves the accuracy

of safe set prediction. The higher accuracy that SCOPE has in predicting safe configurations leads SCOPE to achieve 1.38× lower violation rate than SCOPE-NO (Figure 6-1).

- SCOPE has 1.35× lower violation magnitude than SCOPE-NO for all unsafe configurations in the safe set. This suggests that even when an unsafe configuration from SCOPE is chosen for reconfiguration, it will be likely to generate lower magnitude violation than that from SCOPE-NO. This is reflected in Figure 6-2 where SCOPE has the lowest overall violation magnitudes.

Table 6.1: Summarized results of the percentage of configurations (POC) selected in the total configuration space, coverage, and the average violation magnitudes (AVM) of unsafe configurations in the safe set for SCOPE-NO and SCOPE when the application runs at the last time interval. Higher coverage is better. Lower AVM is better.

	SCOPE-NO			SCOPE		
	POC (%)	Coverage (%)	AVM	POC (%)	Coverage (%)	AVM
als	95.78	38.48	1.47	19.13	71.08	1.16
bayes	95.71	39.77	1.45	19.52	76.73	1.09
gbt	95.66	27.97	1.61	20.81	55.88	1.16
kmeans	94.28	24.85	1.66	18.31	56.04	1.19
linear	95.30	43.24	1.42	20.08	76.32	1.12
lr	95.74	31.73	1.52	18.60	65.47	1.10
nweight	95.30	31.63	1.55	19.87	60.14	1.16
pagerank	95.83	39.92	1.47	20.56	71.41	1.11
pca	93.06	30.19	1.58	18.63	65.60	1.11
rf	92.21	19.13	1.75	17.86	40.69	1.19
terasort	95.55	34.87	1.51	19.04	74.32	1.12
wordcount	95.66	39.83	1.47	18.74	73.61	1.19
Mean	95.01	33.47	1.54	19.26	65.61	1.14

6.4 RQ4: How sensitive is SCOPE to γ?

The operator parameter γ controls the size of neighborhood space that SCOPE explores (Eq. 4.1). To understand the effects of γ on SCOPE, I conduct sensitivity analysis of γ on violation rate and speedup. For better visualization, I normalize each hardware parameter by mean normalization such that each parameter is within range $[-0.5, 0.5]$ [73]. Figure 6-8 shows violation rate and speedup as a function of γ, where the x-axis is the different values of γ and the y-axis is violation rate and speedup. Note that when $\gamma = 0$, this is equivalent to the Static approach since SCOPE must always choose the initial safe configuration. In particular, we observe the following:

- There is a tradeoff between violation rate and speedup; lower violation rate is likely to have lower speedup and higher violation rate is likely to have higher speedup, where ideally, SCOPE would have low violation rate and high speedup. For example, the average violation rate over all applications decreases from 3.04% (when $\gamma = 0.5$) to 0.93% (when $\gamma = 1$), while average speedup of 1.75 (when $\gamma = 0.5$) also decreases to 1.60 (when $\gamma = 1$). The tradeoff occurs because usually when a violation occurs, higher power is consumed, and more work is performed, allowing applications to finish earlier.
- The violation rate is generally higher when $0 < \gamma < 1$, compared to when $\gamma \geq 1$. This is because I normalize all hardware parameters to range from -0.5 to 0.5. Given binary parameters such as hyperthreading (on, off) and number of sockets (1, 2), SCOPE needs γ to be at least 1 to consider updating those parameters. When $0 < \gamma < 1$, SCOPE is not enabled to change hyperthreading or number of sockets at all, which limits the search space for SCOPE to explore safely.

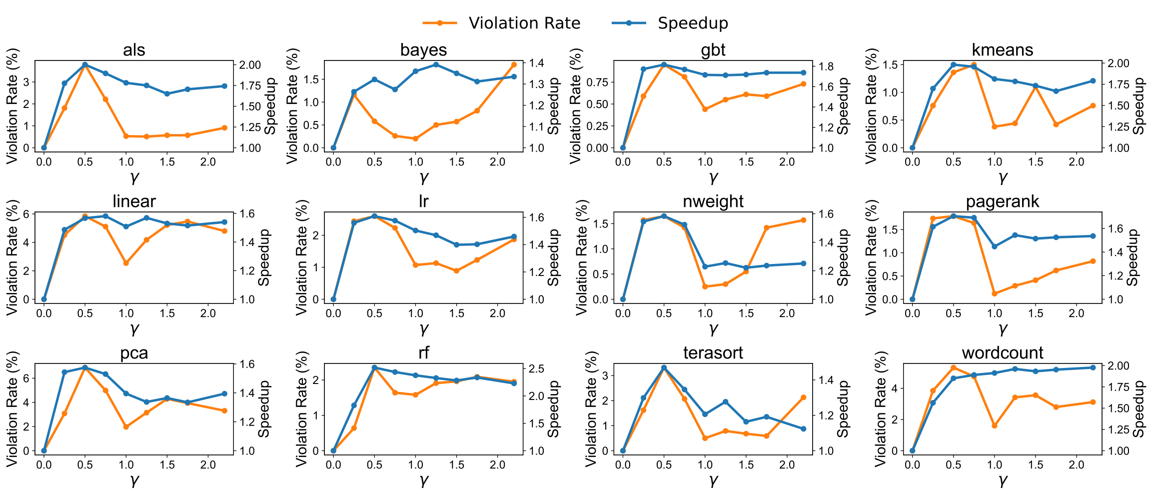

Figure 6-8: Violation rate and speedup as a function of γ averaged over all power constraints for each application. Lower is better for violation rate. Higher is better for speedup.

6.5 RQ5: How sensitive is SCOPE to time intervals?

For all approaches that reconfigure dynamically, I evaluate how they are affected by different time intervals (i.e., the time period between reallocations). I sweep over different time intervals $[5, 10, 20, 30]$ seconds and choose the best time interval (i.e. the interval with the lowest violation rates across all power constraints) for each method and each application.

Figure 6-9 summarizes the violation rate over all power constraints and applications as a function of time interval for each approach, where the x-axis is the different time intervals, and the y-axis is the violation rate.

Compared to Offline, BO, and StageOPT, SCOPE has both the lowest violation rate and the smallest variance at all intervals (SCOPE-NO is the second best and very close to SCOPE). It is because SCOPE and SCOPE-NO construct safe set and optimize latency together for each iteration, while BO optimizes for latency only, and StageOPT separates expanding safe set and optimizing latency into two separate stages. Critically, StageOPT only constructs the safe set at the beginning of execution, so if a previously safe configuration becomes unsafe, then StageOPT has no way to react. SCOPE is even better than SCOPE-NO because it utilizes the locality preserving operator to reconfigure more safely, which helps reduce the overall violation rate. These results show that the novel safe exploration framework of SCOPE is robust to the reconfiguring time intervals.

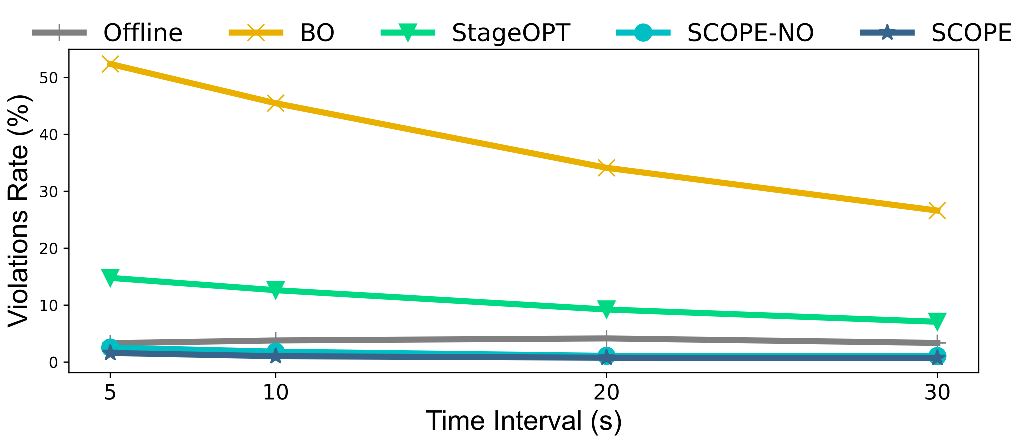

Figure 6-9: Violation rate as a function of time interval averaged over all power constraints and applications for each approach.

6.6 RQ6: How do different types of models perform?

SCOPE is a general framework compatible with any type of learning model. Although Gaussian process regression is chosen as the learning models in all evaluations, I test the framework's generality by using 4 common models: multi-layer perceptron [11, 74] (SCOPE-MLP), linear regression [15] (SCOPE-LINEAR), random forest [51, 52, 50] (SCOPE-RF), and Gaussian process regression [48, 49] (SCOPE). Note that for this experiment, SCOPE uses the same type of models for training both the safety and objective models. Figure 6-10 summarizes the average violation rate and speedup, where the x-axis is the application and the y-axis is either the average violation rate or speedup, with the last column Mean being the arithmetic mean over all applications. We find that:

- SCOPE has the lowest violation rate compared to other learning models. On average, SCOPE has 4.54× lower violation rate than SCOPE-MLP, 10.82× lower violation rate than SCOPE-LINEAR and 12.0× lower violation rate than SCOPE-RF.
- The speedup from using different learning models is relatively similar. SCOPE achieves 1.43× speedup over SCOPE-MLP. SCOPE-LINEAR and SCOPE-RF are faster than SCOPE by 1% and 7% respectively. However, they are not good for being safe since both SCOPE-LINEAR and SCOPE-RF have significantly higher violations than SCOPE.

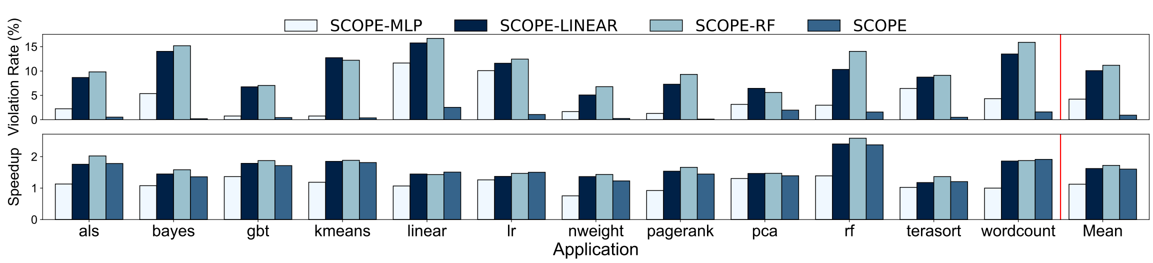

Figure 6-10: Violation rate and speedup of SCOPE using different types of learning models, averaged over all power constraints for each application. Lower is better for violation rate. Higher is better for speedup.

6.7 RQ7: What are the overheads?

This section evaluates the overhead of processing each sample, which includes updating the learning models to predict future system behavior and choosing a new configuration. Figure 6-11 shows the average overhead of processing each sample for each application by different approaches, and the last column Mean is the arithmetic mean over all applications. Static, Oracle and RAPL approaches were omitted; Static and Oracle do not process samples or reconfigure and RAPL uses Intel's power control system to tune parameters in the background and thus the overhead is not measurable. In particular, the results show that:

- BO has the lowest overhead of 0.03s. This is because it only trains one objective model without considering safety.
- SCOPE has the second lowest average overhead and is better than SCOPE-NO because of the locality preserving operator that reduces the number of configurations for inference in predicting safe set.
- The overhead of SCOPE, which is 0.05s on average, is negligible given the total execution time. This is validated by SCOPE achieving lowest latency even though all experiments include overheads. These overheads could be further reduced in future work by porting the resource manager to a lower-level language.

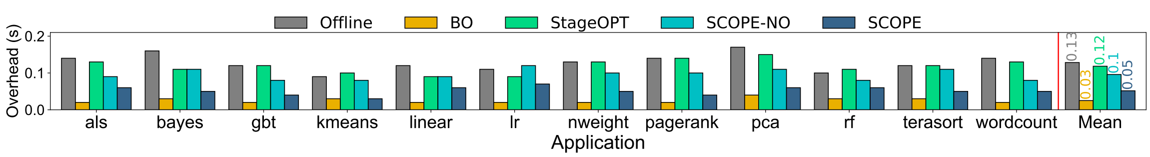

Figure 6-11: Overhead of processing each sample for per reconfiguration averaged over all power constraints for each application. Lower is better.

6.8 Sensitivity Analysis for Offline Approach

The Offline approach that is compared against SCOPE in section 5.3 samples 50% of all possible hardware resource allocations from the target application and input for training. This section details how Offline is affected by the number of samples used. I sweep over different amount of samples, [25, 50, 75, 100]-% of all possible hardware resource allocations using the target application and input and choose the best amount of samples (i.e., the amount that has the lowest violation rate across all power constraints).

Figure 6-12 summarizes the average violation rate and speedup, where the x-axis is the application, the y-axis is the violation rate or speedup, the last column Mean is the arithmetic mean over all applications, and I put the number on each bar to quantify the mean results. I observe that using higher amount of samples has increased speedup for Offline. However, using higher amount of samples does not necessarily lead to lower violation rate, further demonstrating the difficulty of generalization of the a priori collected samples.

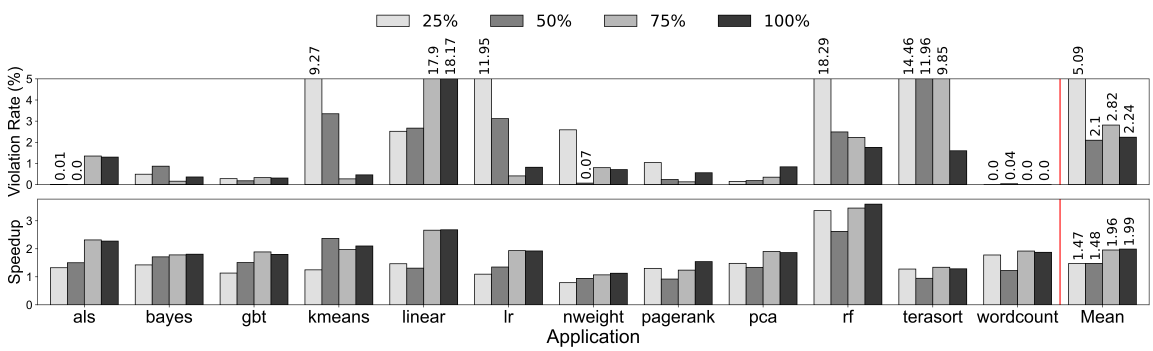

Figure 6-12: Violation rates and speedup of Offline using different amounts of samples, averaged over all power constraint for each application. Lower is better for violation rate. Higher is better for speedup.

Chapter 7

Limitations

I note the following limitations for this work:

- SCOPE, despite achieving the lowest violation rates and magnitudes, does not predict the complete safe set or provide any formal guarantees. Future work can explore formal guarantee for constructing the safe set.
- Although Gaussian process regression provides confidence interval for prediction, such information was not used in SCOPE. Future work can explore techniques for incorporating uncertainty for safe exploration.
- SCOPE currently uses fixed values for parameters such as time interval and the operator parameter γ throughout execution. Future work can explore adaptively changing these parameters during execution to further reducing safety violations.

Chapter 8

Conclusion

This thesis presents SCOPE, a resource manager that leverages a novel safe exploration framework that dynamically allocate hardware resources in the execution space. SCOPE introduces a locality preserving operator that reduces the violation rate and magnitudes compared to prior work. I hope this work can inspire the computer systems community to build new optimization tools that aid in exploration while preserving safety.

Bibliography

[1] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvms. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 232–244, 2017.

[2] Amir M Rahmani, Bryan Donyanavard, Tiago Mück, Kasra Moazzemi, Axel Jantsch, Onur Mutlu, and Nikil Dutt. Spectr: Formal supervisory control and coordination for many-core systems resource management. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 169–183, 2018.

[3] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[4] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. *ACM SIGOPS Operating Systems Review*, 37(5): 149–163, 2003.

[5] Songchun Fan, Seyed Majid Zahedi, and Benjamin C Lee. The computational sprinting game. *ACM SIGARCH Computer Architecture News*, 44(2):561–575, 2016.

[6] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. Caloree: Learning control for predictable latency and low energy. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'18, pages 184–198, New York, NY, USA, 2018. ISBN 9781450349116.

[7] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No" power" struggles: coordinated multi-level power management for the data center. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 48–59, 2008.

[8] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, et al. Data center power oversubscription with a medium voltage

power plane and priority-aware capping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 497–511, 2020.

[9] Shaohong Li, Xi Wang, Faria Kalim, Xiao Zhang, Sangeetha Abdu Jyothi, Karan Grover, Vasileios Kontorinis, Nina Narodytska, Owolabi Legunsen, Sreekumar Kodakara, et al. Thunderbolt:{Throughput-Optimized},{Quality-of-Service-Aware} power capping at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1241–1255, 2020.

[10] Faraz Ahmad and TN Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. *ACM Sigplan Notices*, 45(3):243–256, 2010.

[11] Engin Ïpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. *ACM SIGOPS Operating Systems Review*, 40(5):195–206, 2006.

[12] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 267–281, New York, NY, USA, 2015. ISBN 9781450328357.

[13] Henry Hoffmann. Jouleguard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 198–214, 2015.

[14] Yi Ding, Nikita Mishra, and Henry Hoffmann. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 39–52, 2019.

[15] Yi Ding, Ahsan Pervaiz, Michael Carbin, and Henry Hoffmann. Generalizable and interpretable learning for configuration extrapolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 728–740, 2021.

[16] Zhibin Yu, Zhendong Bei, and Xuehai Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 564–577.

[17] Yanan Sui, Alkis Gotovos, Joel Burdick, and Andreas Krause. Safe exploration for optimization with gaussian processes. In *International conference on machine learning*, pages 997–1005. PMLR, 2015.

[18] Yanan Sui, Vincent Zhuang, Joel Burdick, and Yisong Yue. Stagewise safe bayesian optimization with gaussian processes. In *International conference on machine learning*, pages 4781–4789. PMLR, 2018.

[19] Matteo Turchetta, Felix Berkenkamp, and Andreas Krause. Safe exploration for interactive machine learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[20] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[21] Apache spark. `https://spark.apache.org/docs/2.2.3/configuration.html`.

[22] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194. IEEE, 2010.

[23] Drew D Penney and Lizhong Chen. A survey of machine learning applied to computer architecture design. *arXiv preprint arXiv:1909.12373*, 2019.

[24] Yiying Zhang and Yutong Huang. "learned": Operating systems. *Operating Systems Review*, 53(1):40–45, 2019.

[25] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014. doi: 10.1145/2997641.

[26] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS operating systems review*, 40(5):185–194, 2006.

[27] Joshua J Yi, David J Lilja, and Douglas M Hawkins. A statistically rigorous approach for improving simulation methodology. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 281–291. IEEE, 2003.

[28] Ramazan Bitirgen, Engin Ipek, and Jose F Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 318–329. IEEE, 2008.

[29] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 85–96. IEEE, 2011.

[30] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. Modeling program resource demand using inherent program characteristics. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):1–12, 2011.

[31] Ryan Cochran, Can Hankendi, Ayse K Coskun, and Sherief Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185. IEEE, 2011.

[32] Adam J Oliner, Anand P Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, 2013.

[33] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 91–100, 2012.

[34] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 90–101. IEEE, 2001.

[35] Srinath Sridharan, Gagan Gupta, and Gurindar S Sohi. Holistic run-time parallelism management for time and energy efficiency. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 337–348, 2013.

[36] Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A Jiménez. Bit-level perceptron prediction for indirect branches. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–38. IEEE, 2019.

[37] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V Gratz, and Daniel A Jiménez. Perceptron-based prefetch filtering. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2019.

[38] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425, 2019.

[39] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 861–873, 2021.

[40] Benjamin C Lee, Jamison Collins, Hong Wang, and David Brooks. Cpr: Composable performance regression for scalable multiprocessor models. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 270–281. IEEE, 2008.

[41] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 77–88, New York, NY, USA, 2013. ISBN 9781450318709.

[42] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, pages 127–144, New York, NY, USA, 2014. ISBN 9781450323055.

[43] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA'08, pages 39–50, USA, 2008. ISBN 9780769531748.

[44] Paula Petrica, Adam M Izraelevitz, David H Albonesi, and Christine A Shoemaker. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 13–23, 2013.

[45] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[46] Anthony Canino, Yu David Liu, and Hidehiko Masuhara. Stochastic energy optimization for mobile gps applications. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 703–713, 2018.

[47] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[48] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 469–482, USA, 2017. USENIX Association. ISBN 9781931971379.

[49] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206, 2020. doi: 10.1109/HPCA47549.2020.00025.

[50] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Bliss: autotuning complex applications using a pool of diverse lightweight learning models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1280–1295, 2021.

[51] Luigi Nardi, David Koeplinger, and Kunle Olukotun. Practical design space exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 347–358. IEEE, 2019.

[52] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209. IEEE, 2021.

[53] Jian Li and Jose F Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 77–87. IEEE, 2006.

[54] Christophe Dubach, Timothy M Jones, Edwin V Bonilla, and Michael FP O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 485–496. IEEE, 2010.

[55] Yosef S Razin and Karen M Feigh. Hitting the road: Exploring human-robot trust for self-driving vehicles. In *2020 IEEE International Conference on Human-Machine Systems (ICHMS)*, pages 1–6. IEEE, 2020.

[56] Lukas Brunke, Melissa Greeff, Adam W Hall, Zhaocong Yuan, Siqi Zhou, Jacopo Panerati, and Angela P Schoellig. Safe learning in robotics: From learning-based control to safe reinforcement learning. *Annual Review of Control, Robotics, and Autonomous Systems*, 5, 2021.

[57] Javier Garcia and Fernando Fernández. Safe exploration of state and action spaces in reinforcement learning. *Journal of Artificial Intelligence Research*, 45:515–564, 2012.

[58] Hongzi Mao, Malte Schwarzkopf, Hao He, and Mohammad Alizadeh. Towards safe online reinforcement learning in computer systems. In *Workshop on ML for Systems In conjunction with 33rd conference on neural information processing systems*, 2019.

[59] Matteo Turchetta, Andrey Kolobov, Shital Shah, Andreas Krause, and Alekh Agarwal. Safe reinforcement learning via curriculum induction. *Advances in Neural Information Processing Systems*, 33:12151–12162, 2020.

[60] Akifumi Wachi and Yanan Sui. Safe reinforcement learning in constrained markov decision processes. In *International Conference on Machine Learning*, pages 9797–9806. PMLR, 2020.

[61] Wanqiao Xu, Kan Xu, Hamsa Bastani, and Osbert Bastani. Safely bridging offline and online reinforcement learning. *arXiv preprint arXiv:2110.13060*, 2021.

[62] Huazhe Zhang and H Hoffman. A quantitative evaluation of the rapl power control system. *Feedback Computing*, 6, 2015.

[63] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi, and Jinquan Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*, pages 41–51, 2010.

[64] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[65] Olivier Temam. A defect-tolerant accelerator for emerging high-performance applications. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 356–367. IEEE, 2012.

[66] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278. IEEE, 2016.

[67] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nature*, 585:357–362, 2020.

[68] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[69] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[70] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. *ACM SIGPLAN Notices*, 51(4):545–559, 2016.

[71] Subho S Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Bayesperf: minimizing performance monitoring errors using bayesian statistics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 832–844, 2021.

[72] Yuzhao Wang, Lele Li, You Wu, Junqing Yu, Zhibin Yu, and Xuehai Qian. Tpshare: a time-space sharing scheduling abstraction for shared cloud via vertical labels. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 499–512. IEEE, 2019.

[73] Md Jahangir Alam, Pierre Ouellet, Patrick Kenny, and Douglas O'Shaughnessy. Comparative evaluation of feature normalization techniques for speaker verification. In

International Conference on Nonlinear Speech Processing, pages 246–253. Springer, 2011.

[74] Engin Ipek, Bronis R De Supinski, Martin Schulz, and Sally A McKee. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*, pages 196–205. Springer, 2005.